Relazione Programmatore JTAG

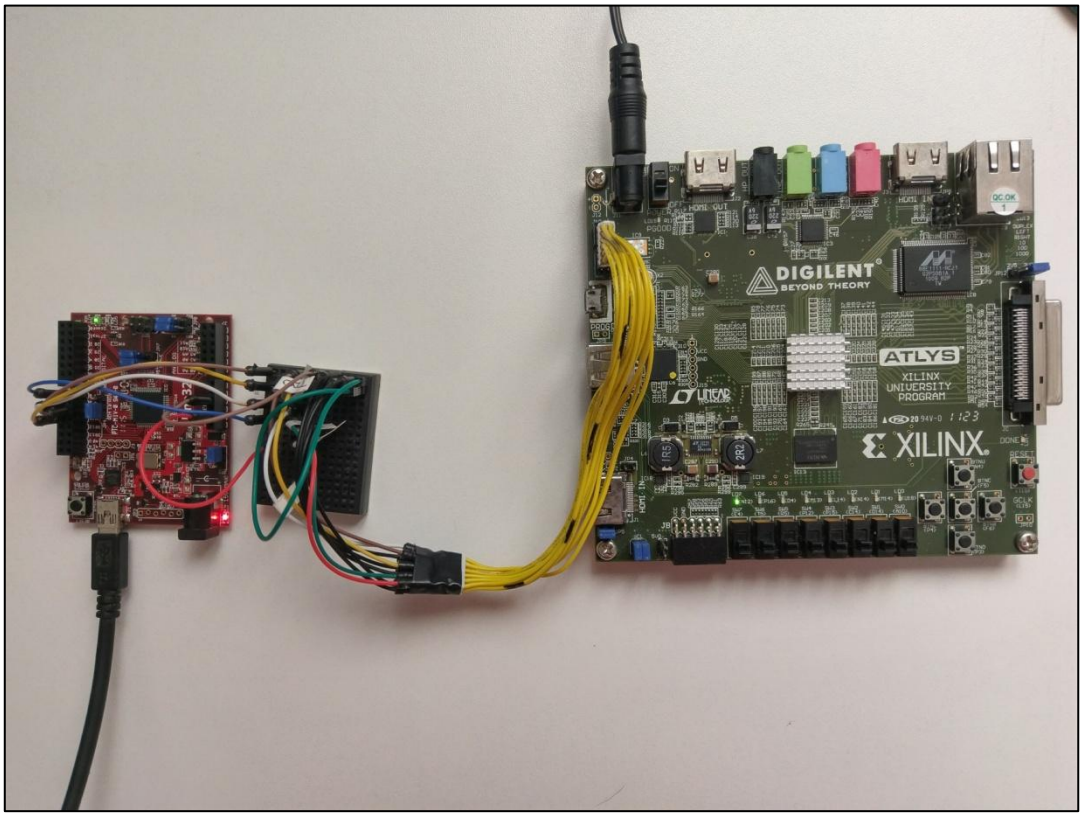

Figura 1: Componenti del progetto

Progetto

L'obiettivo del progetto è quello di sostituire il programmatore USB della scheda Xilinx Spartan 6 con una board Arduino-like e un software da noi realizzato per effettuare, tramite protocollo JTAG, la programmazione della scheda FPGA.

Componenti del gruppo:

- Dal Ben Mattia (Matricola: 102806)
- Marson Mattia (Matricola: 103248)
- Guglielmini Manuel (Matricola: 91575)

File che vanno a costituire il progetto:

- JTAG.ino
- serial.hpp, serial.cpp
- svftoserial.cpp svftoserial.hpp
- main.cpp
- blink_led.svf

Componenti

Gli elementi hardware e software che vanno a realizzare il programmatore sono schematizzati in figura.

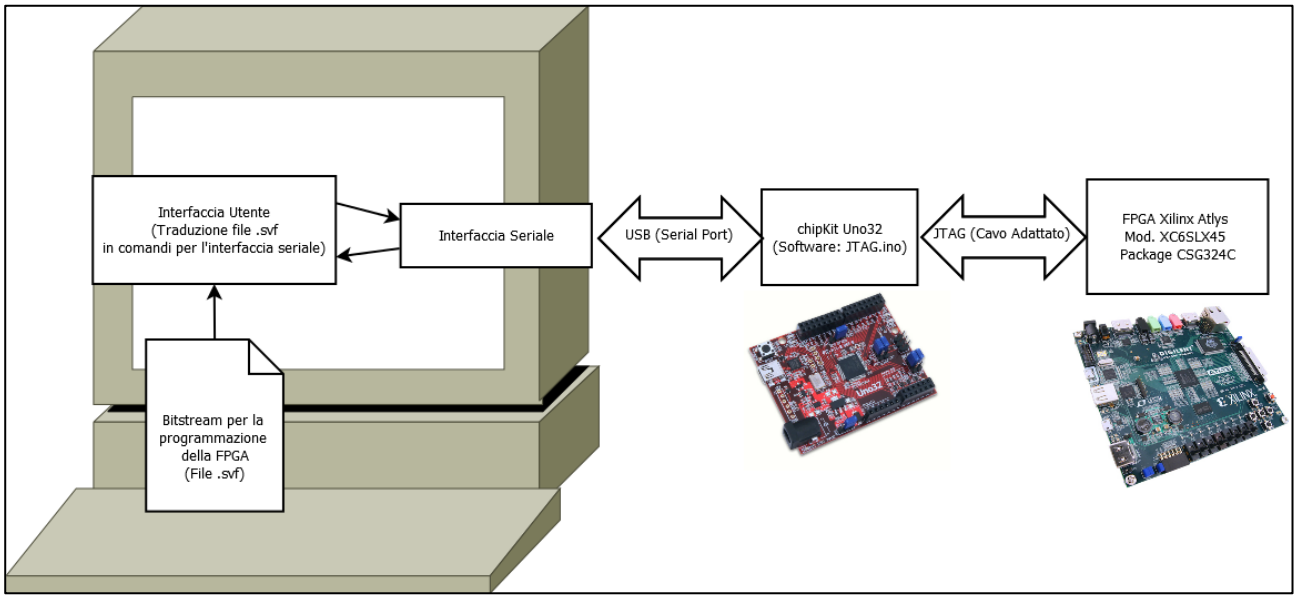

Figura 2: Schematizzazione dei componenti

Passiamo quindi a descrivere brevemente i componenti che sono stati realizzati nell'ambito del progetto:

- **Cavo JTAG**: il cavo è stato realizzato per permettere la connessione tra la scheda Uno32 e l' Header JTAG della FPGA (J10) ed ha previsto lo studio dei datasheet della scheda e lo standard JTAG.
- **JTAG.ino**: è il software che genera i segnali da inviare all'header JTAG della scheda target e traduce i comandi inviati sulla porta seriale in segnali elettrici.
- **Interfaccia Seriale**: permette l'invio dei comandi da PC verso la scheda Uno32. Questa parte è implementata da serial.hpp e serial.cpp.
- **Interfaccia Utente**: raccoglie i comandi immessi dall'utente e li invia alla scheda Uno32, inoltre si fa carico della traduzione dei file SVF nella corrispondente sequenza di comandi JTAG. Questa parte è implementata da main.cpp, svftoserial.cpp e svftoserial.hpp.
- **Bitstream per la programmazione della FPGA (blink_led.svf)** : per effettuare i test e verificare il corretto funzionamento del software e dell'hardware si è generato un bitstream di programmazione semplice. Nel nostro caso si tratta di un semplice lampeggiamento di un led.

Protocollo JTAG

Prima di procedere con l'analisi dettagliata dei singoli componenti è opportuno analizzare il protocollo su cui si basa il progetto. **JTAG**, acronimo di **J**oint **T**est **A**ction **G**roup, è un consorzio di 200 imprese produttrici di circuiti integrati creato allo scopo di definire un protocollo standard per il test funzionale di tali dispositivi. Questo consorzio ha dato vita a quello che poi è diventato lo standard **IEEE 1149.1** noto colloquialmente come standard JTAG. Siccome la connessione JTAG costituisce una porta di accesso riservata, viene utilizzata anche per la programmazione dei singoli dispositivi. Il progetto qui trattato sfrutta quest'ultima funzionalità.

Tutte le operazioni dello standard JTAG sono controllate attraverso la **T**est **A**ccess **P**ort (TAP) del dispositivo. La porta TAP consta di quattro segnali:

- **TCK:** (Test Clock) Segnale di clock dei dati
- **TMS:** (Test Mode Select) Selezione dello stato
- **TDI:** (Test Data In) Segnale di ingresso dei dati del dispositivo
- **TDO:** (Test Data Out) Segnale di uscita dei dati

Questi segnali interagiscono con il TAP Controller: una macchina a stati finiti a 16 stati riportata qui di seguito.

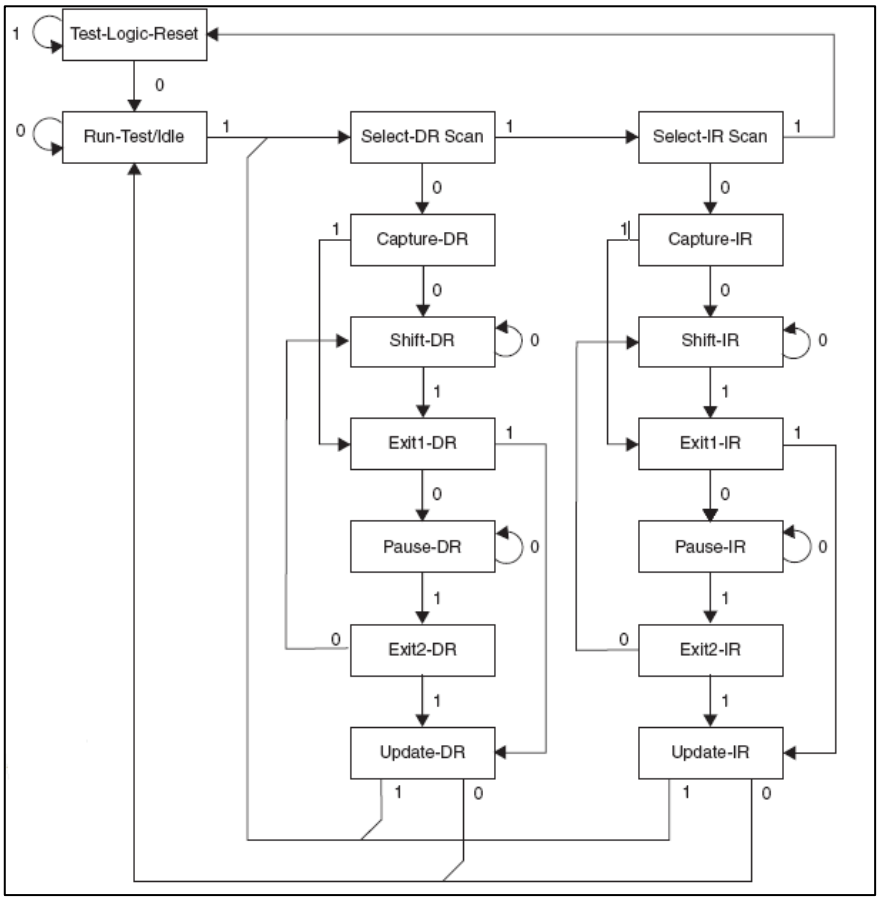

Figura 3: Diagramma degli stati del JTAG TAP Controller

Lo standard IEEE 1149.1 definisce il comportamento della macchina a stati TAP e l'attività dei pin in uscita in accordo con specificate attività sui pin di ingresso. La transizione degli stati avviene sul fronte di salita del segnale TCK. Il valore dell'ingresso TMS determina quale sarà lo stato successivo nella macchina a seguito della transizione. Il valore dell'ingresso TDI viene campionato sul fronte di salita del segnale TCK quando la macchina si trova negli stati SHIFT-IR o SHIFT-DR. L'uscita TDO è pilotata solamente quando il TAP sta shiftando dati o istruzioni, essa inizia a generare output quando il TAP si porta negli stati SHIFT-IR o SHIFT-DR.

Una sequenza di stati TAP degna di nota è quella che garantisce che la macchina si trovi nello stato Test-Logic-Reset (TLR). A partire da qualunque stato della macchina, mantenere TMS alto per almeno cinque cicli di TCK, mette la macchina nello stato TLR.

Tutte le operazioni JTAG shiftano dati dentro o fuori dai registri dati o dal registro istruzioni. Il controller TAP garantisce l'accesso a questi registri. Vi sono due classi di registri JTAG:

- Registro Istruzioni: il cui accesso è dato dallo stato SHIFT-IR
- Registri Dati: il cui accesso è dato dallo stato SHIFT-DR

Per shiftare dati attraverso questi registri, il TAP controller del dispositivo target deve essere impostato al corrispondente stato della macchina. Ad esempio, per scrivere dei dati all'interno del Registro Istruzioni, è necessario portare la macchina nello stato SHIFT-IR e quindi shiftare i dati a partire dal LSB.

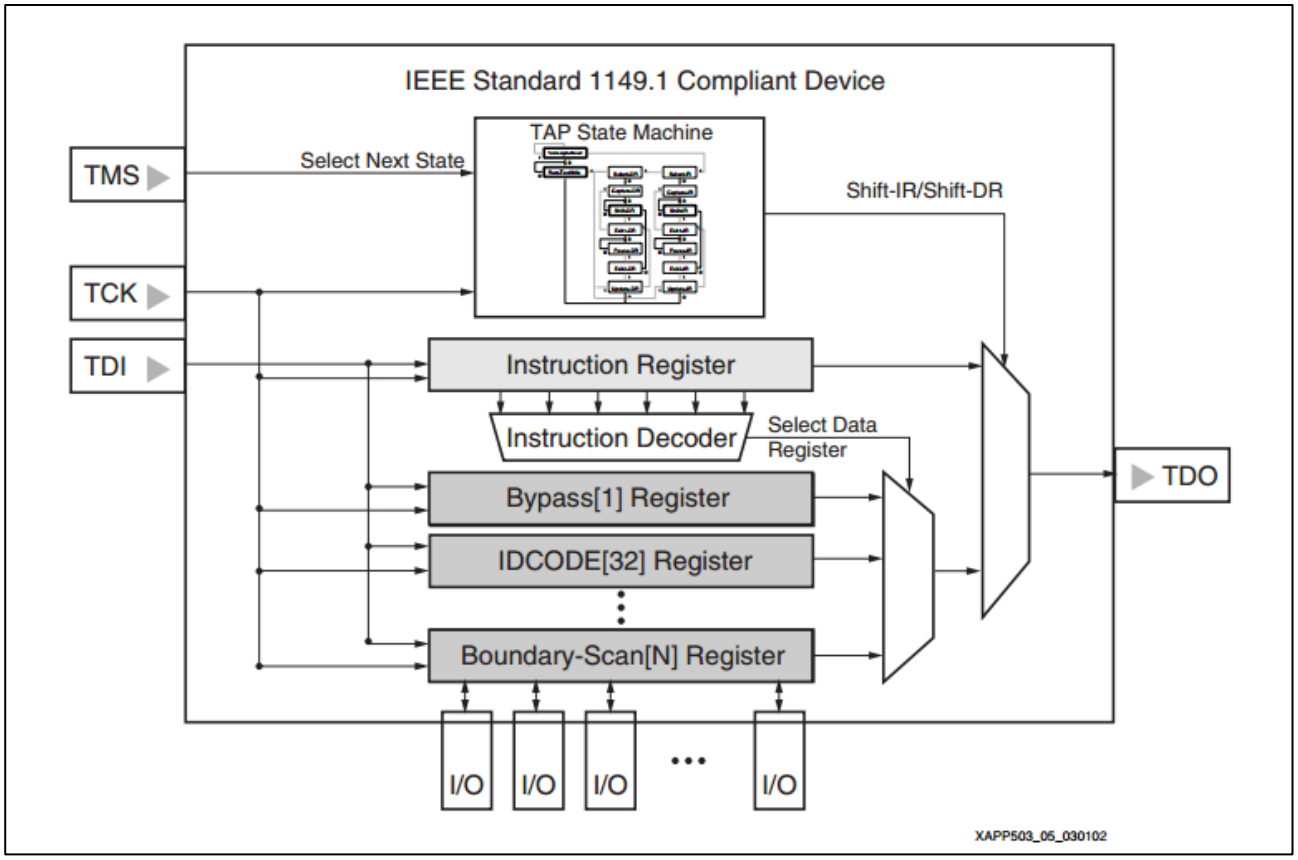

Figura 4: Architettura JTAG tipica

Esempio per la comprensione:

La sequenza di comandi necessaria per poter leggere l'ID code del dispositivo target è la seguente.

TAP Controller step e descrizione	TDI	TMS	TCK
Stato TLR	X	1	5
Stato Run Test/Idle	X	0	1
Stato SELECT-IR	X	1	2
Stato SHIFT-IR	X	0	2
Carichiamo l'istruzione in memoria partendo dal LSB	10010	0	5
L'ultimo bit va scritto entrando nello stato EXIT1-IR	0	1	1
Stato UPDATE-IR	X	1	1
Stato RTI	X	0	1
Stato SELECT-DR	X	1	1
Stato SHIFT-DR	X	0	1
Shift dei bit	00000...	0	31
L'ultimo bit va scritto entrando nello stato EXIT1-DR	0	1	1
Stato UPDATE-DR	X	1	1
Stato RTI	X	0	1

Per convenzione si è usato il carattere "X" per indicare che lo stato del segnale non è importante ai fini del funzionamento del protocollo (don't care).

Immettendo questa sequenza di segnali, quando ci troviamo nello stato SHIFT-DR, vedremo in uscita sul TDO passare l'ID code del dispositivo. Per convenzione questo uscirà partendo dal LSB, per leggerlo correttamente sarà quindi necessario leggerlo al contrario.

Si noti che questa sequenza di istruzioni è valida solamente per le schede della serie Spartan 6. Altri dispositivi posseggono codici istruzioni diversi per poter leggere l'ID code. Inoltre è possibile che la dimensione del registro istruzioni cambi a seconda della tipologia della scheda: in questo caso abbiamo un registro istruzioni da 6 bit ma altre schede posseggono registri da 8 o 5 bit.

Realizzazione del progetto

L'approccio allo sviluppo del progetto è stato di tipo bottom-to-top: inizialmente abbiamo studiato lo standard JTAG, quindi abbiamo realizzato il software per la board Uno32. In seguito abbiamo costruito il cavo per la connessione con l'header JTAG e effettuato i primi test di funzionamento. Da qui si è partiti per generare il codice per la comunicazione tramite porta seriale tra PC e board Uno32. Infine, dopo aver studiato il relativo standard, si è andati a realizzare il software di interfaccia verso l'utente il quale inoltre traduce i file SVF in comandi JTAG.

Analizziamo ora i componenti del progetto in modo più approfondito.

Cavo JTAG

Per realizzare il collegamento tra la scheda Uno32 e la FPGA è stato necessario costruire un cavo apposito basandosi sullo standard JTAG ed i datasheet della scheda Xilinx.

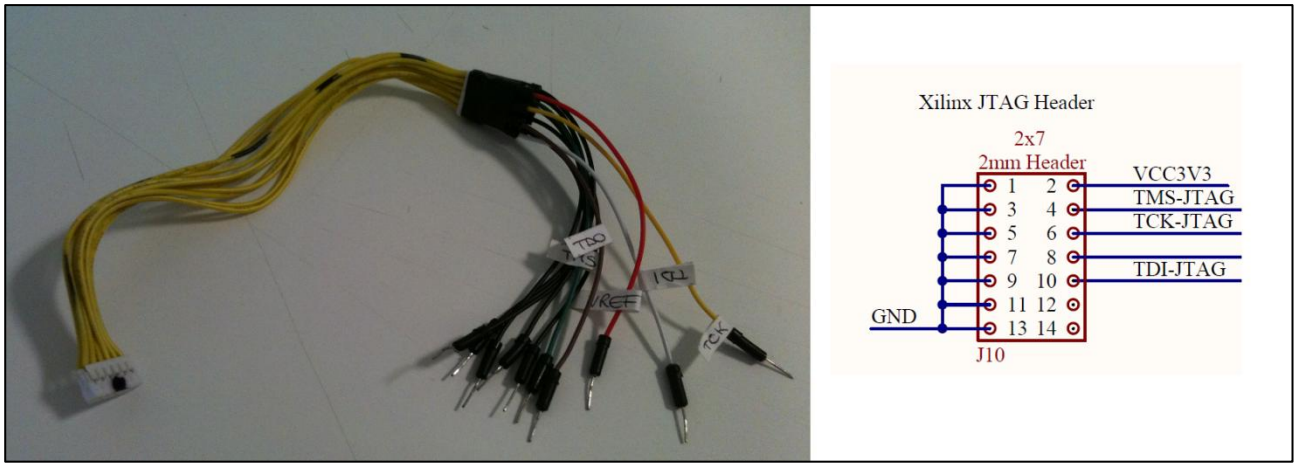

Figura 5: Cavo realizzato e header JTAG

Il cavo è composto da due connettori a 12 pin a cui, ad una estremità, sono stati collegati dei jumper da breadboard di tipo maschio/maschio. Questo è stato necessario per poter collegare fisicamente la scheda Uno32 dotata di connettori femmina da 0,1" e la scheda Xilinx dotata di header da 14 pin. Si noti che la mancanza di 2 pin nel cavo non compromette la funzionalità poichè 4 dei 14 pin dell'header JTAG sono inutilizzati.

Il pin 8, non specificato in figura, è il pin riservato al segnale JTAG TDO

Schematico dei collegamenti

Qui di seguito è riportato lo schema delle connessioni realizzate dal cavo, da noi creato, tra board Uno32 e la FPGA.

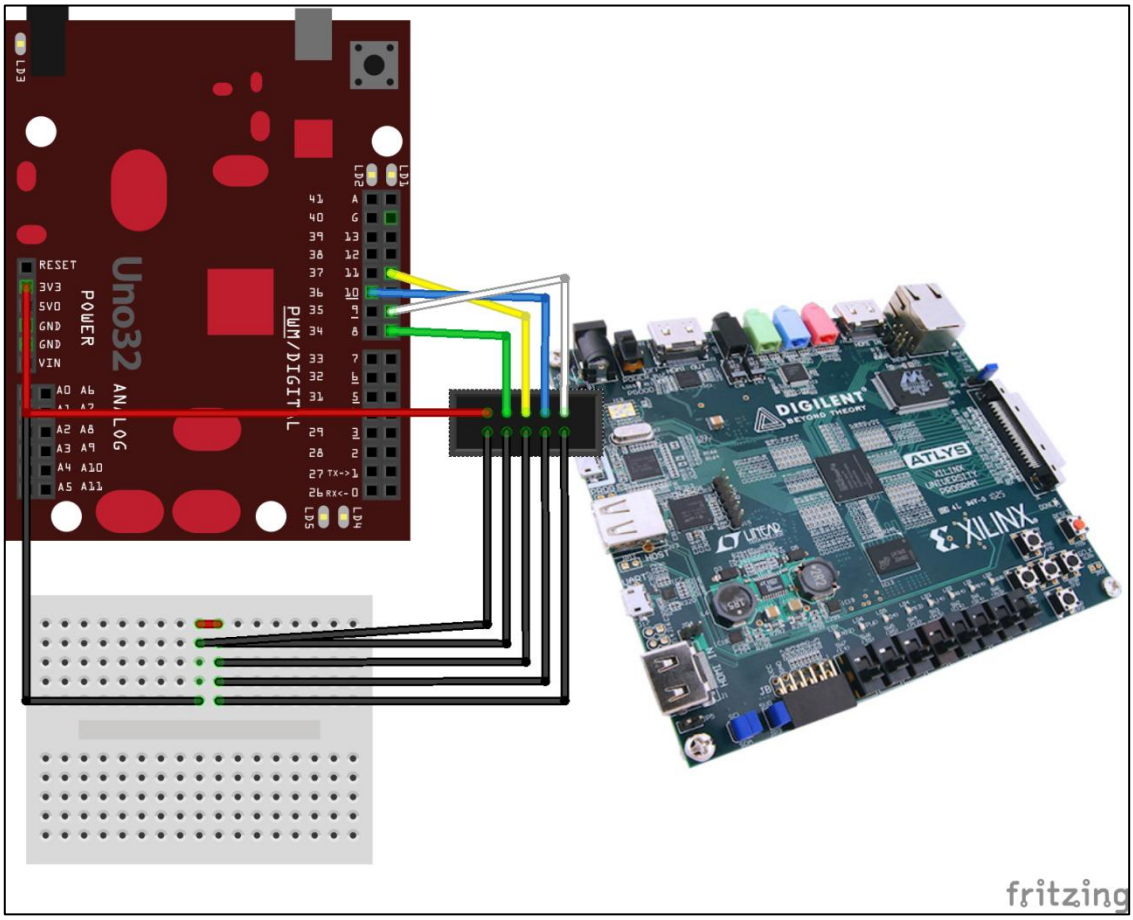

Figura 6: Schematico dei Collegamenti

JTAG.ino

Andiamo quindi ad analizzare il codice che viene eseguito dalla board Uno32.

Il codice permette l'invio degli impulsi elettrici necessari al corretto funzionamento della macchina a stati JTAG per mezzo di semplici comandi inviati tramite porta seriale.

I segnali che ci servono per comunicare con la macchina a stati JTAG, come visto in precedenza, sono solamente quattro: TMS, TCK, TDI e TDO.

I comandi implementati nel codice sono fondamentalmente sette e sono tutti quelli necessari al funzionamento del programma:

- **Comando "!"**: TMS = 1 , quindi viene eseguito un colpo di clock (TCK va a 1, aspetta 25 µs quindi torna a zero).
- **Comando "*"**: TMS = 0, quindi viene eseguito un colpo di clock.
- **Comando "."**: Viene impostato TDI = 0, si esegue un colpo di clock, infine il programma va a leggere il bit che passa su TDO.

- **Comando ","**: Viene impostato TDI = 1, si esegue un colpo di clock, infine il programma va a leggere il bit che passa su TDO.
- **Comando ":"**: TMS = 1, TDI = 0, colpo di clock e va a leggere il bit che passa su TDO.
- **Comando ";"**: TMS = 1, TDI = 1, colpo di clock e va a leggere il bit che passa su TDO.
- **Comando "[0-f]"**: vengono scritti 4 bit di seguito in formato esadecimale e letti altrettanti bit che transitano su TDO. In pratica è un loop di 4 cicli che effettua una scrittura di TDI, un colpo di clock e una lettura di TDO.
- **Comando "X"**: Viene eseguita la sequenza completa per andare a leggere l'ID Code della scheda FPGA, nonché specchia i bit in uscita della scheda perché venga letto correttamente. Sfrutta i comandi citati in precedenza.

Si noti che i comandi "**:**" e "**;**" sono molto specializzati in quanto vengono utilizzati solamente quando è necessario uscire dallo stato SHIFT-IR o SHIFT-DR. Lo standard prevede infatti che per l'uscita dal loop di shift dei registri sia necessario inviare l'ultimo bit di scrittura del registro contemporaneamente all'uscita dello stato. Ciò si traduce quindi in due operazioni contemporanee:

1) La scrittura di TMS a 1 per uscire dallo stato di SHIFT.
2) La scrittura di TDI per l'inserimento dell'ultimo bit da immettere nel registro.

Il comando "**X**" invece è stato utilizzato soprattutto in fase di debug del codice, in quanto la cattura dell'ID code (noto per mezzo dei programmi "ufficiali") ci ha permesso di valutarne l'effettivo funzionamento.

Funzionamento

JTAG.ino si basa interamente su un vettore di char chiamato buf (buffer) di dimensione BUFSIZE. Esso ad ogni immissione dei comandi viene riempito dei comandi descritti nel paragrafo precedente. Essi vengono ricevuti sulla porta seriale, letti in maniera distruttiva, quindi interpretati dal codice e tradotti nei segnali elettrici corrispondenti. Quindi i bit di risposta emessi dalla FPGA vengono letti e interpretati dalla board Uno32 che li salva, così come le arrivano, nello stesso vettore di char utilizzato in precedenza e inviati indietro tramite serial port per poter essere visualizzati sul monitor del computer collegato alla board.

Questo riutilizzo del buffer è stato necessario per ridurre al minimo il consumo delle risorse, già scarse, di una board Arduino. Nel nostro caso però, si aveva a disposizione una board dalle risorse molto meno limitate e quindi non sarebbe necessario questo accorgimento. Si è deciso di mantenere questa scelta per permettere uno sviluppo del codice più parallelo in quanto avevamo a disposizione una sola board Uno32 ma tutti i componenti del gruppo avevano accesso ad un Arduino.

I comandi descritti in precedenza sono quindi mappati sulle seguenti funzioni:

- Scrittura di TMS: `void tms(boolean value);`
- Scrittura di TDI e lettura di TDO: `boolean tdi(boolean value);`
- Scrittura multipla di TDI e lettura di TDO: `uint8_t tdin(int n, uint8_t bits);`
- Colpo di clock (Pulse di TCK): `void tck(void);`
- Operazioni uscita dal loop di shifting: `boolean exit_r(boolean value);`
- Lettura ID code: `void idcode(void);`

Esempio per la comprensione:

Riferendoci all'esempio riportato a pagina 5 vediamo come viene tradotto sfruttando il codice di JTAG.ino.

TAP Controller step e descrizione	TDI	TMS	TCK	Istruzione JTAG.ino
Stato TLR	X	1	5	**!!!!!**
Stato Run Test/Idle	X	0	1	*****
Stato SELECT-IR	X	1	2	**!!**
Stato SHIFT-IR	X	0	2	****
Carichiamo l'istruzione in memoria partendo dal LSB	10010	0	5	**,..,.**
L'ultimo bit va scritto entrando nello stato EXIT1-IR	0	1	1	**:**
Stato UPDATE-IR	X	1	1	**!**
Stato RTI	X	0	1	*****
Stato SELECT-DR	X	1	1	**!**
Stato SHIFT-DR	X	0	1	*****
Shift dei bit	00000...	0	31	**0000000...**
L'ultimo bit va scritto entrando nello stato EXIT1-DR	0	1	1	**:**
Stato UPDATE-DR	X	1	1	**!**
Stato RTI	X	0	1	*****

A seguito dell'immissione di questi comandi la board risponderà stampando sulla seriale il valore del ID code. Si può verificare il corretto funzionamento del codice immettendo direttamente sulla seriale il comando "**!!!!!*!!**,..,.:!*!*0000000...:!***" sfruttando l'IDE di Arduino. Sulla finestra comparirà l'ID code del dispositivo che nel nostro caso è 34008093 (si tenga presente che verrà stampato a partire dal LSB quindi è necessario invertire i numeri per ricavare il codice identificativo corretto).

Interfaccia seriale

L'interfaccia seriale realizzata in questo ambito si occupa semplicemente di trasferire i caratteri immessi sul PC verso la board Uno32. Si è deciso di realizzarla in quanto ci si voleva svincolare dall'utilizzo del software della board Uno32 che altrimenti avrebbe reso molto più difficile lo sviluppo del software di traduzione dei file di tipo SVF.

Metodi principali

Vediamo ora brevemente i metodi principali implementati nell'interfaccia seriale:

- `void Serial::Open(const char *Device,const unsigned Bauds)`
  Apre la comunicazione sulla porta seriale tramite la porta specificata, utilizzando il baud rate passato come secondo argomento.
- `void Serial::Close()`
  Chiude la comunicazione sulla seriale.
- `void Serial::WriteChar(const char Byte)`
  Scrive sulla seriale il carattere passato come argomento.
- `void Serial::WriteString(const char *String)`
  Scrive sulla seriale la stringa passata come argomento.
- `void Serial::Write(const void *Buffer, const unsigned NbBytes)`
  Scrive sulla seriale il buffer di caratteri passato come primo argomento; il numero di caratteri del buffer da scrivere è specificato tramite il secondo argomento del metodo.
- `unsigned Serial::ReadChar(char *pByte, unsigned TimeOut_ms)`
  Legge un carattere dalla seriale. È possibile specificare un tempo di timeout.
- `unsigned Serial::ReadStringNoTimeOut(char *String, char FinalChar, unsigned MaxNbBytes)`
  Legge una stringa dalla seriale, restituita tramite il puntatore specificato come argomento. Inoltre va specificato il carattere di terminazione (secondo argomento) e il numero massimo di caratteri da leggere.
- `unsigned Serial::ReadString(char *String,char FinalChar,unsigned MaxNbBytes,unsigned TimeOut_ms)`
  Legge una stringa dalla seriale, restituita tramite il puntatore specificato come argomento. Inoltre va specificato il carattere di terminazione (secondo argomento), il numero massimo di caratteri da leggere e il timeout per la lettura
- `unsigned Serial::Read (void *Buffer,unsigned MaxNbBytes,unsigned TimeOut_ms)`
  Legge dalla seriale un buffer di caratteri, restituito tramite il puntatore passato come primo argomento. Inoltre va indicato il numero massimo di caratteri da leggere (primo argomento) e il timeout per la lettura.
- `void Serial::FlushReceiver()`
  Svuota il buffer di lettura, scartando i caratteri non letti.
- `int Serial::Peek()`
  Restituisce il numero di caratteri nel buffer di lettura, quindi non ancora letti. La classe TimeOut è una classe che racchiude i metodi necessari per la corretta gestione dei timeout
- `void TimeOut::InitTimer()`
  Inizializza il timer.
- `unsigned long int TimeOut::ElapsedTime_ms()`
  Restituisce il tempo trascorso dall'inizializzazione del timer. Il tempo trascorso è espresso in millisecondi.

Standard SVF

A questo punto è necessario parlare dello standard alla base dei file SVF necessari per programmare la scheda Xilinx. Le informazione riportate e riassunte nel seguito sono state ricavate dall'Application Note 503 v2.1 della Xilinx (link: https://www.xilinx.com/support/documentation/application_notes/xapp503.pdf).

Serial **V**ector **F**ormat (**SVF**) è un formato di file contenente boundary scan vectors che devono essere inviati ad un circuito elettronico sfruttando l'interfaccia JTAG. I boundary scan vectors consistono dei seguenti dati:

- **Stimulus data**: Questi sono i dati che devono essere inviati al dispositivo
- **Risposta attesa**: Questi sono i dati che ci si aspetta il dispositivo ritorni se non ci sono errori
- **Mask data**: Definisce quali bit della risposta vanno considerati validi e quali vanno ignorati
- Informazioni addizionali su come inviare i dati (esempio: massima frequenza di clock)

Un file SVF è definito per mezzo di un file ASCII consistente di un insieme di asserzioni SVF. Ogni asserzione consiste di un comando e dei parametri ad esso associati. Lo standard SVF specifica molteplici comandi ma per il nostro progetto sono stati usati i quattro comandi fondamentali.

Scan Instruction Register (SIR)

```
SIR length TDI (tdi) SMASK (smask) [TDO (tdo) MASK (mask)];
```

dove:

- **length**: specifica il numero di bit che devono essere shiftati nello stato SHIFT-IR.
- **TDI**: specifica lo scan pattern da applicare allo stato SHIFT-IR.
- **SMASK**: specifica quali bit ignorare nello scan pattern (1 = care, 0 = don't care).
- **TDO**: specifica che output ci aspettiamo shiftando i dati nello stato SHIFT-IR.
- **MASK**: specifica quali bit ignorare nell'uscita attesa del TDO (1 = care, 0 = don't care).

Scan Data Register (SDR)

```
SDR length TDI (tdi) SMASK (smask) [TDO (tdo) MASK (mask)];
```

dove:

- **length**: specifica il numero di bit che devono essere shiftati nello stato SHIFT-DR.
- **TDI**: specifica lo scan pattern da applicare allo stato SHIFT-DR.
- **SMASK**: specifica quali bit ignorare nello scan pattern (1 = care, 0 = don't care).
- **TDO**: specifica che output ci aspettiamo shiftando i dati nello stato SHIFT-DR.
- **MASK**: specifica quali bit ignorare nell'uscita attesa del TDO (1 = care, 0 = don't care).

RUNTEST

```
RUNTEST run_count TCK;
```

L'istruzione RUNTEST specifica un ammontare di tempo per il quale il TAP controller aspetta nel stato Run-Test/Idle. Questo tempo di attesa è richiesto dall'algoritmo di programmazione di alcuni dispositivi Xilinx.

run_count: specifica il numero di cicli TCK da aspettare nello stato Run-Test/Idle.

STATE

```
STATE tap_state;
```

tap_state: specifica uno stato della macchina a stati del JTAG al quale spostarsi. Ad esempio "STATE RESET" impone di spostarsi nello stato Test Logic Reset.

Esempio per la comprensione:

Qui di seguito è riportato un esempio in cui sono elencate le istruzioni necessarie a leggere l'ID code di una scheda e i corrispondenti effetti sulla macchina a stati del protocollo JTAG.

TAP Controller step e descrizione	TDI	TMS	TCK	Istruzione SVF
Stato Test Logic Reset	X	1	5	**STATE RESET ;**
Stato Run Test/Idle	X	0	1	**STATE IDLE ;**
Stato SELECT-IR	X	1	2	**SIR 6 TDI (09) ;**
Stato SHIFT-IR	X	0	2	
Carichiamo l'istruzione in memoria partendo dal LSB	10010	0	5	
L'ultimo bit va scritto entrando nello stato EXIT1-IR	0	1	1	
Stato UPDATE-IR	X	1	1	
Stato RTI	X	0	1	
Stato SELECT-DR	X	1	1	**SDR 32 TDI (00000000) ;**
Stato SHIFT-DR	X	0	1	
Scriviamo i bit	00000...	0	31	
L'ultimo bit va scritto entrando nello stato EXIT1-DR	0	1	1	
Stato UPDATE-DR	X	1	1	
Stato RTI	X	0	1	

Interfaccia utente

Main.cpp

Questa parte del codice si occupa di caricare in memoria l'intero file SVF e darlo in pasto alla parte di traduzione (svftoserial).

Dopo aver aperto la porta seriale per comunicare con l'Uno32, il programma carica in un vettore di stringhe l'intero file SVF per mezzo della funzione `ExtractInstruction(file_svf)`. Una volta estratte le istruzioni inizia il ciclo for per sottoporle a traduzione una alla volta grazie alla funzione `DecodeInstruction(string)` trattata nel paragrafo seguente. Quando l'istruzione è stata decodificata in modo da poter essere eseguita dal software che si trova sull'Uno32, essa viene scritta sulla seriale e il programma si mette in attesa di ricevere un riscontro dall'Uno32. Qualora l'Uno32 risponda in maniera corretta il programma può procedere con una nuova istruzione. Questo ciclo di attesa è necessario a garantire una sincronizzazione tra il PC e la board Uno32.

Si noti che per il caricamento del bitstream che rappresenta il programma che deve eseguire la FPGA il comportamento è leggermente diverso. Una volta rilevata una istruzione molto lunga per mezzo della funzione `isBitStreamInstruction(string)` ne estraiamo solamente i dati da immettere nello stato SHIFT-DR (funzione `ExtractBitstream(vector<string>)`) contenuto fra le parentesi e questo bitstream lo decodifichiamo per mezzo della funzione `GenerateBITSTREAMOutput(vector<string>)` trattata nel seguito. A questo punto inizia un ciclo per la scrittura del bitstream decodificato immesso riga per riga. Ad ogni invio di una riga abbiamo un ciclo di attesa per dare all'Uno32 il tempo di scrivere i dati sulla scheda. Si noti che ogni carattere della stringa corrisponde a 4 bit per i quali sono necessari 4 colpi di clock da 25µs. Inoltre il bitstream, come da standard JTAG, va immesso a partire dal LSB fino al MSB; la funzione di decodifica si fa carico anche di questa operazione.

Modalità manuale

In fase di debugging è stata implementata una modalità manuale per l'immissione di comandi sulla board Uno32. Facendo partire l'eseguibile con argomento "-m" è possibile inserire direttamente i comandi per l'Uno32 trattati a pagina 7-8.

Svftoserial.cpp

Questa parte del codice si occupa della traduzione del contenuto del file SVF negli opportuni comandi che, previo passaggio tramite interfaccia seriale, saranno utilizzati da JTAG.ino per generare i corrispondenti segnali d'uscita.

La funzione principale del programma è

```
string DecodeInstruction (string line)
```

che assume di ricevere in ingresso una stringa rappresentante una riga di testo del file SVF. Essa riconoscerà il tipo di comando inviato in base all'analisi del primo campo della stringa e passerà la gestione a una delle funzioni specifiche analizzate in seguito.

Analizziamo ora le funzioni specifiche per ogni comando:

- `string GenerateSDROutput (string line)`
  Si occupa di generare sulla stringa restituita l'output corrispondente all'istruzione SDR ricevuta; in particolare, visto che i dati veri e propri devono essere inviati a partire dal bit meno significativo, si occupa di rovesciare il contenuto della stringa in ingresso.
- `string GenerateSIROutput (string line)`
  Restituisce una stringa con l'output corrispondente all'istruzione SIR ricevuta; anche qui l'istruzione è inserita a partire dal LSB e, dato che è passata in esadecimale, viene convertita in binario e ne vengono presi solo il numero di bit opportuni, pari alla dimensione del relativo registro.
- `string GenerateSTATEOutput (string line)`
  Fornisce l'uscita opportuna per le istruzioni di tipo STATE, in particolare i comandi per raggiungere lo stato RESET o IDLE, a seconda dei casi.
- `string GenerateRUNTESTOutput (string line)`
  Restituisce una stringa con zeri in numero tale da riuscire a produrre i cicli di clock richiesti dal secondo campo di questa istruzione.

Come indicato nel file SVF specifico "blink_led.svf" si è assunto che lo stato a cui si passa al termine delle istruzioni SIR e SDR sia quello di IDLE (è quello indicato dalle istruzioni ENDIR e ENDDR, che specificano quanto appena scritto).

Istruzioni quali TIR, HIR, TDR e HDR si occupano di gestire più FPGA collegate in Daisy-Chain e non vengono prese in considerazione dalla funzione principale in quanto si è sviluppato il progetto ipotizzando di lavorare con un solo dispositivo. Questa scelta è dovuta al fatto che avere più FPGA da programmare contemporaneamente avrebbe complicato di molto il progetto e il tempo necessario per lo sviluppo sarebbe stato troppo elevato. Anche altre istruzioni che non producono output non sono state prese in considerazione da tale funzione ma sono state implementate opportunamente (si veda ENDIR e ENDDR). I commenti all'interno del file SVF vengono scartati in fase di estrazione (ExtractInstruction).

Un discorso a parte va fatto per la funzione `GenerateBITSTREAMOutput(vector<string>)` la quale prende in ingresso l'intero bitstream da caricare sulla scheda e lo manipola per poter essere inviato correttamente dall'Uno32. Essa esegue le seguenti operazioni:

- Elimina dall'istruzione i bit della SMASK.
- Specchia l'intero bitstream in modo che venga inserito partendo dal LSB fino al MSB (quindi sulla seriale viene scritto a partire dall'ultima riga fino alla prima e ogni riga viene scritta a partire dall'ultimo carattere fino al primo).
- Traduce gli ultimi quattro bit in modo da sfruttare il comando "**:**" o "**;**" all'occorrenza.

Si è deciso di realizzare una funzione a parte per scrivere il bitstream in quanto la sua lunghezza necessitava un comportamento differenziato rispetto al resto delle istruzioni.

Sequenza di istruzioni per ricavare l'ID code di un dispositivo

Per semplificare la comprensione di quanto viene effettuato dal sistema implementato in questo progetto riprendiamo l'esempio utilizzato in precedenza.

Il seguente file SVF va a leggere l'ID code della scheda:

```
1  STATE RESET ;
2  STATE IDLE ;
3  SIR 6 TDI (09) SMASK (3f) ;
4  SDR 32 TDI (00000000) SMASK (ffffffff) TDO (f4008093) MASK (0fffffff) ;
```

Il programma andrà a tradurre nel seguente modo:

Istruzione SVF	TAP Controller step e descrizione	TDI	TMS	TCK	Istruzione per l'Uno32
STATE RESET ;	Stato TLR	X	1	5	**!!!!!**
STATE IDLE ;	Stato Run Test/Idle	X	0	1	*****
SIR 6 TDI (09) ;	Stato SELECT-IR	X	1	2	**!!**
	Stato SHIFT-IR	X	0	2	******
	Carichiamo l'istruzione in memoria partendo dal LSB	10010	0	5	**,..,.**
	L'ultimo bit va scritto entrando nello stato EXIT1-IR	0	1	1	**:**
	Stato UPDATE-IR	X	1	1	**!**
	Stato RTI	X	0	1	*****
SDR 32 TDI (00000000) ;	Stato SELECT-DR	X	1	1	**!**
	Stato SHIFT-DR	X	0	1	*****
	Shift dei bit	00000...	0	31	**0000000...**
	L'ultimo bit va scritto entrando nello stato EXIT1-DR	0	1	1	**:**
	Stato UPDATE-DR	X	1	1	**!**
	Stato RTI	X	0	1	*****

Manuale del software

Vediamo ora brevemente come utilizzare il software prodotto nella realizzazione di questo progetto.

Modalità di esecuzione

Il software dispone di quattro modalità di esecuzione:

```
prompt> ./interface
```

Esecuzione senza argomenti. Modalità di esecuzione standard. Prevede il minimo output su terminale e quindi, durante l'upload del bitstream, verrà visualizzato solamente lo stato dell'avanzamento.

```
prompt> ./interface -m
```

Esecuzione con argomento "-m". Modalità di esecuzione manuale, descritta in precedenza, atta ad inserire direttamente i comandi trattati a pagina 7-8.

```
prompt> ./interface -v
```

Esecuzione con argomento "-v". Modalità di esecuzione con output dettagliato. Durante l'upload del bitstream verrà visualizzata la risposta dell'Uno32 per ogni riga inviata attraverso la porta seriale.

```
prompt> ./interface -f [fileName]
```

Esecuzione con argomento "-f". Modalità di esecuzione con output su file. Le risposte generate dall'Uno32 verranno raccolte e salvate su file di testo. Se non viene specificato il nome del file, l'output verrà riportato nel file: "arduino_log.txt".

Schermata di aiuto

```
prompt> ./interface -h
```

È inoltre possibile richiamare su schermo un piccolo riassunto delle modalità disponibili per mezzo dell'argomento "-h".

Significato output sul terminale

Passiamo ora a spiegare il significato dell'output sul terminale. Si faccia riferimento alla seguente figura.

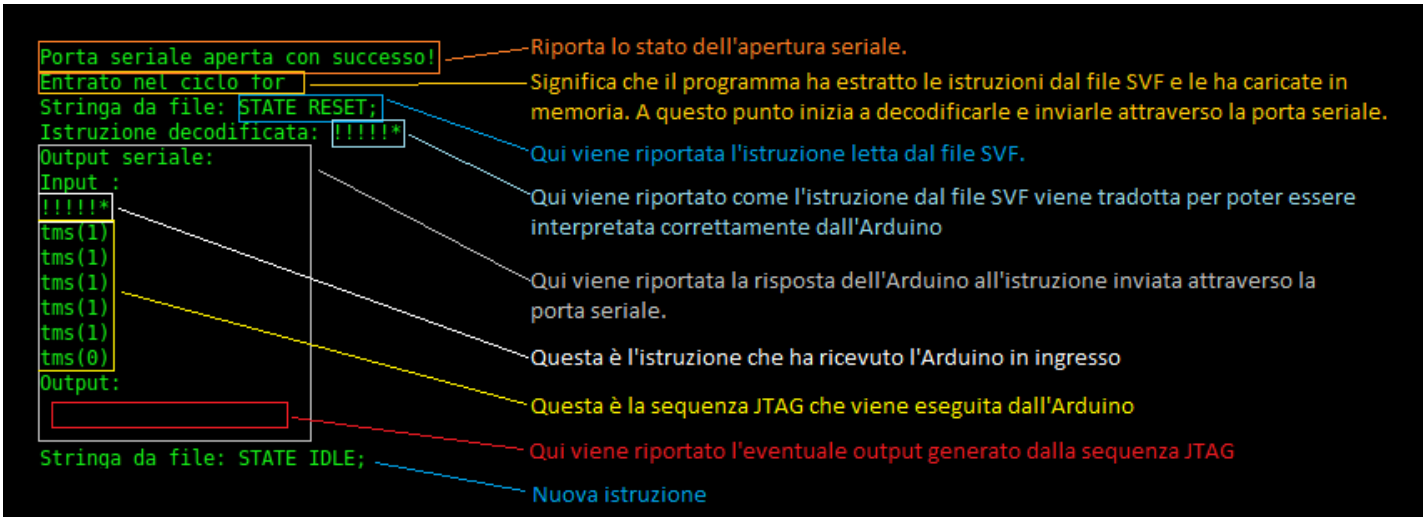

Figura 7: Output terminale per la modalità di esecuzione standard.

Valutazione sperimentale

Il principale problema riscontrato nel software è relativo ai tempi di esecuzione. La prima versione del programma prevedeva un tempo di upload del bitstream di circa 20 ore. Ciò era dovuto ad una scelta sbagliata della lunghezza delle stringhe da inviare sulla seriale e ad un baud rate troppo basso. Usare stringhe di 250 caratteri e un baud rate di 115200 baud (il massimo supportato dall'Uno32) ci ha permesso di abbassare il tempo di esecuzione fino a 7 ore.

A seguito di questa operazione il collo di bottiglia si è rivelato essere il periodo di TCK. È stato quindi ridotto da 1ms a 5µs e si arrivati ad un tempo di upload di 17 minuti.

Si può già notare come il tempo di esecuzione non dipenda linearmente dal segnale di clock. Ciò è dovuto all'elaborazione in fase di decodifica delle istruzioni e alle operazioni di I/O sulla seriale che introducono ritardi.

Sono stati stimati i seguenti tempi di esecuzione in fase di test.

	Periodo TCK 1ms	**Periodo TCK 5µs**
9600 baud	548 minuti (≅9 ore)	160 minuti (≅2.5 ore)
115200 baud	409 minuti (≅7 ore)	17 minuti

Mantenendo il baud rate a 115200 baud invece sono state effettuate le seguenti misure:

Periodo TCK	**Tempo necessario all'upload del bitstream**
1 ms	409 minuti
100 µs	54 minuti
10 µs	19 minuti
1 µs	15 minuti

Quindi, perché sia possibile abbassare ulteriormente i tempi di esecuzione del programma, è necessario svolgere un attento lavoro di ottimizzazione sulla comunicazione seriale e sul software dell'Arduino.

Conclusioni

Infine possiamo affermare che il progetto è stato concluso con successo in quanto siamo riusciti a programmare correttamente la FPGA che ci è stata fornita.

Possibili miglioramenti e ulteriori spunti per lo sviluppo del progetto potrebbero essere:

- Sostituire la scheda chipKit Uno32 con una scheda più performante per poter scrivere più velocemente il bitstream sulla scheda target.
- Sperimentare fino a che punto è possibile abbassare il ritardo tra fronti di clock nel software Arduino per velocizzare l'upload del bitstream. Allo stato attuale l'upload del bitstream è l'operazione che occupa più tempo: per il file blink_led.svf utilizzato nei nostri test l'operazione occupa circa 20 minuti.
- Testare il funzionamento su schede diverse. Il programma da noi sviluppato è stato testato esclusivamente su una scheda Xilinx 6SLX45. Poiché, come specificato nella documentazione della Xilinx, il comportamento del dispositivo cambia radicalmente da scheda a scheda, è possibile che il programma non funzioni su dispositivi diversi da quella presa in esame. Durante la realizzazione del progetto si è tenuto conto di questo aspetto non facendo assunzioni sulla architettura JTAG della scheda target; ciononostante non è possibile garantire il funzionamento su altre FPGA senza prima procedere con ulteriori test.
- Testare il funzionamento con diversi file SVF.